Logistics

Tonight:
   HW3 @ 11:59 pm

Monday:
   Exercise 15 @ 10:30 am
Casting in C++

Four different casts that are more explicit:

1. `static_cast<to_type>(expression)`
2. `dynamic_cast<to_type>(expression)`
3. `const_cast<to_type>(expression)`
4. `reinterpret_cast<to_type>(expression)`

When programming in C++, you should use these casts!
**Static Cast**

`static_cast<to_type>(expression)`

**Used to:**

1) Convert pointers of related types
   
   ```cpp
   Base* b = static_cast<Base*>(new Derived);
   ``
   
   - compiler error if types aren't related

2) Non-pointer conversion

   ```cpp
   int qt = static_cast<int>(3.14);
   ```
**Static Cast**

`static_cast<to_type>(expression)`

[[!] Be careful when casting up:

```cpp
Derived* d = static_cast<Derived*>(new Base);
d->y = 5;
```
- compiler will let you do this
- dangerous if you want to do things defined in Derived, but not in Base!
Dynamic Cast

dynamic_cast<to_type>(expression)

Used to:

1) Convert pointers of related types

   Base* b = dynamic_cast<Base*>(new Derived);
   - compiler error if types aren't related
   - at runtime, returns nullptr if it is actually an unsafe upwards cast:

   Derived* d = dynamic_cast<Derived*>(new Base);
Const Cast

\texttt{const\_cast<to\_type>}(expression)

Used to:
1) Add or remove const-ness
   \begin{verbatim}
   const int x = 5;
   const int *ro_ptr = &x
   int *ptr = \texttt{const\_cast<int*>}(ro_ptr);
   \end{verbatim}
Reinterpret Cast

`reinterpret_cast<to_type>(expression)`

Used to:
1) Cast between incompatible types
   ```
   int* ptr = 0xDEADBEEF;
   int64_t x = reinterpret_cast<int64_t>(ptr);
   - types must be of same size
   - refuses to do float-integer conversions
   ```
Exercise 1
# C++ Code Snippet

```cpp
class Base {
public:
    int x;
};

class Derived : public Base {
public:
    int y;
};

int64_t x = 0x7fffffffffe870;
char* str = reinterpret_cast<char*>(x);

void foo(Base *b) {
    Derived *d = dynamic_cast<Derived*>(b);
    // additional code omitted
}

Derived *d = new Derived;
Base *b = static_cast<Base*>(d);

double x = 64.382;
int64_t y = static_cast<int64_t>(x);
```
Computer Networks: A 7-ish Layer Cake
Computer Networks: A 7-ish Layer Cake

0101 →

bit encoding at signal level

physical → physical → physical
Computer Networks: A 7-ish Layer Cake

- Format/meaning of messages
- Sending data end-to-end
- Routing of packets across networks
- Multiple computers on a local network
- Bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

Routing of packets across networks
Multiple computers on a local network
Bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- Sending data end-to-end
- Routing of packets across networks
- Multiple computers on a local network
- Bit encoding at signal level
Computer Networks: A 7-ish Layer Cake

- format/meaning of messages
- sending data end-to-end
- routing of packets across networks
- multiple computers on a local network
- bit encoding at signal level

HTTP  DNS

application  application
presentation  presentation
session  session
transport  transport
network  network
data link  data link
physical  physical
Data flow

Transmit Data

Receive Data

application
presentation
session
transport
network
data link
physical

application
presentation
session
transport
network
data link
physical
Exercise 2
Exercise 2

- format/meaning of messages
- sending data end-to-end
- routing of packets across networks
- multiple computers on a local network
- bit encoding at signal level
Exercise 2

- **DNS:** Translating between IP addresses and host names. (Application Layer)
- **IP:** Routing packets across the Internet. (Network Layer)
- **TCP:** Reliable, stream-based networking on top of IP. (Transport Layer)
- **UDP:** Unreliable, packet-based networking on top of IP. (Transport Layer)
- **HTTP:** Sending websites and data over the Internet. (Application Layer)
TCP versus UDP

Transmission Control Protocol (TCP)
- Connection oriented Service
- Reliable and Ordered
- Flow control

User Datagram Protocol (UDP)
- Connectionless service
- Unreliable packet delivery
- Faster
- No feedback
Sockets

- Just a file descriptor for network communication
- Types of Sockets
  - Stream sockets (TCP)
  - Datagram sockets (UDP)
- Each socket is associated with a **port number** and an **IP address**
  - Both port and address are stored in network byte order (big endian)

```c
struct sockaddr_in:

+---------+--------+-------+-------+-------+-------+
| family  | port   | addr  |     zero |
+---------+--------+-------+-------+-------+-------+
| 0       | 2      | 4     | 8     | 16    |
```

```c
struct sockaddr_in6:

+---------+--------+-------+-------+-------+-------+-------+-------+-------+-------+
| fam     | port   | flow  |     addr     |     scope |
+---------+--------+-------+-------+-------+-------+-------+-------+-------+-------+
| 0       | 2      | 4     | 8     | 24    | 28    |
```
# Sockets

**struct sockaddr** (pointer to this struct is used as parameter type in system calls)

<table>
<thead>
<tr>
<th>fam</th>
<th>???</th>
<th>???</th>
<th>???</th>
</tr>
</thead>
</table>

**struct sockaddr_in** (IPv4)

<table>
<thead>
<tr>
<th>fam</th>
<th>port</th>
<th>addr</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

**struct sockaddr_in6** (IPv6)

<table>
<thead>
<tr>
<th>fam</th>
<th>port</th>
<th>flow</th>
<th>addr</th>
<th>scope</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
</tr>
</tbody>
</table>

**struct sockaddr_storage**

<table>
<thead>
<tr>
<th>fam</th>
</tr>
</thead>
</table>

Big enough to hold either...
Byte Ordering and Endianness

- **Network Byte Order (Big Endian)**
  - The most significant byte is stored in the highest address

- **Host byte order**
  - Might be big or little endian, depending on the hardware

- To convert between orderings, we can use
  - `uint16_t htons (uint16_t hostlong);`
  - `uint16_t ntohs (uint16_t hostlong);`
  - `uint32_t htonl (uint32_t hostlong);`
  - `uint32_t ntohl (uint32_t hostlong);`
Exercise 3
specify lookup hints

1. ________ (hostname, servname, ___, ___)
1. **getaddrinfo()**

- Performs a **DNS Lookup** for a hostname

```c
int getaddrinfo(const char *hostname,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```
1. `getaddrinfo()`

- Performs a **DNS Lookup** for a hostname
- Use “hints” to specify constraints
- Get back a linked list of `struct addrinfo` results

```c
int getaddrinfo(const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **res);
```
1. **getaddrinfo() - Interpreting Results**

```c
struct addrinfo {  
    int ai_flags;   // additional flags  
    int ai_family;  // AF_INET, AF_INET6, AF_UNSPEC  
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM, 0  
    int ai_protocol; // IPPROTO_TCP, IPPROTO_UDP, 0  
    size_t ai_addrlen; // length of socket addr in bytes  
    struct sockaddr* ai_addr; // pointer to socket addr  
    char* ai_canonname; // canonical name  
    struct addrinfo* ai_next; // can form a linked list  
};
```

- `ai_addr` points to a `struct sockaddr` describing the socket address
1. **getaddrinfo() - Interpreting Results**

With a `struct sockaddr*`:

- The field `sa_family` describes if it is IPv4 or IPv6
- Cast to `struct sockaddr_in*` (v4) or `struct sockaddr_in6*` (v6) to access/modify specific fields
- Store results in a `struct sockaddr_storage` to have a space big enough for either
2.

extract fields from result (IPv4 vs IPv6)

_______

_______(____, type, protocol)

_______
2. socket()

- Creates a “raw” socket, ready to be bound
- Returns file descriptor (sockfd) on success, -1 on failure

```c
int socket(int domain, // AF_INET, AF_INET6
           int type,   // SOCK_STREAM (TCP)
           int protocol); // 0
```
3.

extract fields from result (IPv4 vs IPv6)

```plaintext
int (sa_family)
```

```plaintext
2
________ (____, type, protocol)
```

```plaintext
int (sockfd)
```

```plaintext
3
________ (____, _______, addrlen)
```
3. `connect()`

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure

```
int connect (int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

```
extract fields from result (IPv4 vs IPv6)

int (sa_family)

struct sockaddr_storage*

int (sockfd)

socket(domain, type, protocol)

connect(sockfd, serv_addr, addrlen)
```
3. connect()

- Connects an available socket to a specified address
- Returns 0 on success, -1 on failure

```c
int connect (int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

```c
Cast sockaddr_storage* to sockaddr*!
```
4. read/write and 5. close

- Thanks to the file descriptor abstraction, use as normal!
- read from and write to a buffer, the OS will take care of sending/receiving data across the network
- Make sure to close the fd afterward